York University
Department of Electrical Engineering and Computer Science
VLSI Chip Design EECS 4612

Project #2: Major Project (v4)

1 Specifications

1.1 Overall Functionality

You'll be designing an ASIC capable of carrying out a coordinated set of operations. In particular your system should be able to do the following. It should be able to do a matrix-vector product

$$\mathbf{y} = \mathbf{W}\mathbf{x}. \tag{1}$$

where $\mathbf{W}$ is a $M \times N$ matrix and $\mathbf{x}$ is a $N \times 1$ vector. M and N are integers. $\mathbf{W}$ and $\mathbf{x}$ will be available in a separate memory for your ASIC to fetch (this memory is not part of your ASIC). The computed result $\mathbf{y}$ is a $M \times 1$ vector.

The individual components (i.e., elements) of $\mathbf{W}$, $\mathbf{x}$, and $\mathbf{y}$ can be referred to as $W_{i,j}$, x_j, and y_i, respectively. The i and j symbols denote index integers spanning $[0, M-1]$, and $[0, N-1]$, respectively.

Your ASIC should also be able to run its previously computed elements of $\mathbf{y}$, y_i, through a sub-word selector (SWS)

$$y'_i = \mathrm{SWS}_a^b(y_i) = y_i[\mathtt{b} : \mathtt{a}];\ \mathtt{b} \geq \mathtt{a}. \tag{2}$$

As summarized above, for any binary word y_i fed into it, the SWS returns the sequence of bits (the sub-word) spanning bit position index `a` to `b` of y_i. For clarity, the least significant

© 2023 S. Magierowski

('first') bit is assumed to be at bit position index 0, the 'last' bit in a 32-bit word is assumed to be at bit position index 31.

Your output should also be able to transform the elements your previously computed sub-word y'_i as follows.

$$z_i = \text{ReLU}(y'_i) \tag{3}$$

where the ReLU$(\cdot)$ function (i.e., the "rectifier linear unit") passes positive arguments unchanged, but outputs zero if its argument is negative.

Your ASIC should be capable of supporting a streaming sequence of calculations. That is, you should expect your system to be able to handle matrix/vector input pairs $(\mathbf{W}_0, \mathbf{x}_0)$, $(\mathbf{W}_1, \mathbf{x}_1)$, ... one after the other and produce a sequence of corresponding results. The details of how this is to be handled are discussed in § 1.4.

If your ASIC is asked to just produce $\mathbf{y}$ then in response to a stream of inputs you should produce a stream of outputs $\mathbf{y}_0, \mathbf{y}_1, \ldots$ Or, if it's the sub-word data that is requested, then you should produce a stream of outputs $\mathbf{y'}_0, \mathbf{y'}_1, \ldots$ Or, if it's the rectified results that are requested, then your ASIC should produce the sequence $\mathbf{z}_0, \mathbf{z}_1, \ldots$

Generalizing, we'll denote a matrix/vector input set within a stream using $\mathbf{W}_k$, $\mathbf{x}_k$. That is, this denotes the kth input to handle out of some stream of inputs. Similarly, $\mathbf{y}_k$, $\mathbf{y'}_k$, and $\mathbf{z}_k$ denote the calculation results corresponding to that input. Your ASIC should deal with only one stream set, k, at a time. It should not accept a request for a new stream set until it has completed a stream set that it is working on. When it is safe to refer to either $\mathbf{y}_k$ or $\mathbf{y'}_k$ without needing to distinguishing between them we will refer to these terms with the catch-all symbol $\mathbf{r}_k$.

Please note that in the communications scheme details below, k, actually take on a deeper meaning. It not only contains information about which stream is being processed, but it also contains information about how the data of each stream is distributed in memory. This property of k is detailed further in § 4.1.

1.2 System Constraints

1.2.1 Undergrads

- The M and N can be any number between 1 and 64 inclusive.
- The elements of $\mathbf{W}$ and $\mathbf{x}$ can only be 8-bit 2's complement values.
- The elements of $\mathbf{y}$ should be 32-bit 2's complement values.
- The sub-word fetched by SWS will always be 8-bits long and have $0 \leq \mathtt{a}, \mathtt{b} \leq 31$.
- ReLU assumes that its input is 2's complement and it output is 2's complement.

- You won't be asked to output (to some device outside of the ASIC) the result **y**. We'll only ask for $\mathbf{y}'$ or **z** for any given k.

1.2.2 Grads

You'll need to do at least as much as the undergrads (see bullet list above), but with the following extensions.

- The elements of **W** and **x** may be be 8-bit or 16-bit 2's complement values. You'll be told which one in the *cmd.initiate* instruction (just a bit setting in the opcode field most likely).
- The elements of **y** should be 48-bit 2's complement values.
- The sub-word fetched by SWS may be 8-bits or 16-bits.
- Besides ReLU also implement a tanh (both of which assume 2's complement inputs and produce 2's complement outputs). The output of tanh is referred to as ϕ.

1.3 Input/Output

Your system has four main interfaces to the outside world:

- `cmd`: means for commands to reach your ASIC from an external processor (PROC)
- `resp`: means for your ASIC to send status signals to an external processor (PROC)
- `mem.req`: means for your ASIC to request a load/store from/to memory (MEM)
- `mem.resp`: means for your ASIC to receive load data from memory (MEM)

A high-level picture of how these interfaces link your ASIC to PROC and MEM is shown in Fig. 1. More details about these connections and the way they are operated are given below.

1.4 Operational Logic

The logic of system operation is as follows.

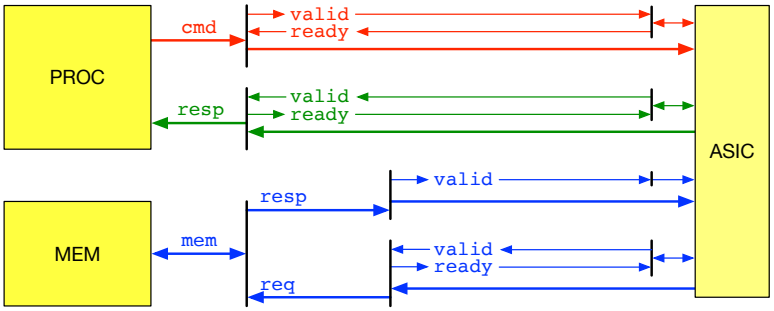

Figure 1: Interfaces between ASIC and PROC & MEM. A high-level overview.

1.4.1 PROC-ASIC Communication

1) PROC initiates your ASIC's calculations via a set of different commands coming into your ASIC through the `cmd` interface. For more concise presentation we'll label these commands with the prefix label *cmd.* This naming scheme is to aid human understanding only. This label is not directly known by the ASIC.

2) If the PROC is interested in having your ASIC complete a new calculation (stream set) k it should first tell your ASIC about this with the *cmd.initiate* command delivered to your ASIC via the `cmd` interface.

3) The PROC's *cmd.initiate* command tells your ASIC that a new calculation is desired and which terms it should return, that is, whether your ASIC should return (all) the elements of $\mathbf{y'}_k$ or (all) the elements of $\mathbf{z}_k$. Only one of these two will be asked for per matrix/vector input (stream set) k. These values are returned on the `mem.resp` port which is discussed later.

4) After *cmd.initiate* a series of commands from PROC to ASIC follow. These commands always arrive in the same order.

5) After *cmd.initiate* PROC will send *cmd.size*, this commands tells your ASIC that the values of M, N, and `a` as well as the stream set value k are available. Specifically, when this command is valid, these will be available to you on the `cmd.rs1` port.

6) After *cmd.size* PROC sends *cmd.addrW*, this command tells your ASIC the address of the first term element of $\mathbf{W}_k$ (i.e., $w_{0,0}^{(k)}$) is available. Specifically, when this command is valid, you should find the address on the `cmd.rs1` port. $\mathbf{W}_k$'s elements are organized in row-major order in main memory (i.e., the row of a matrix are contiguously arranged in PROC's memory, MEM). The system uses byte addressing (i.e., each address denotes the location of a byte in MEM).

7) After *cmd.addrW* PROC sends *cmd.addrX*, this command tells your ASIC the starting address of the first element of the vector $\mathbf{x}_k$. Again, the actual address appears on the `cmd.rs1` port. The elements of $\mathbf{x}_k$ are also arranged contiguously in order from the first element, $x_0^{(k)}$, on.

8) After *cmd.addrX* PROC sends *cmd.addrR*, this command tells your ASIC the starting address of the first element in which to put your $\mathbf{r}_k$ results (whichever one you were asked to output by *cmd.initiate*).

9) After your ASIC confirms that it has registered *cmd.addrR* (the logic of `cmd`'s val/rdy connection confirms this) then PROC assumes your ASIC has started to do the calculation. As before, the actual address for $\mathbf{r}_k$ appears on the `cmd.rs1` port.

10) Once your ASIC has finished sending all of its computed data to memory ('finished' means that you sent the last result byte your ASIC thinks it needs to send, an event confirmed by `mem.resp` val/rdy connection). It should indicate to the PROC that it is done by sending a `resp.data` message (along with `resp.rd` information) that declares the status of the work. Only two things need to be indicated in the response, one is that the job finished successfully and the other is that the process failed (no need to explain the failure is required). More details on the resp message are provided in § 4.2.

11) After it has consumed your *resp.data* message (by asserting *resp.ready*) PROC may send you another *cmd.initiate* message on the `cmd` interface beginning the process of yet another calculation (e.g., for matrix/vector input stream set $k+1$ or some other integer value of PROC's choice). And so on.

1.4.2 ASIC Loads from MEM

The items above outline the main communications procedures between the ASIC and PROC. There is still the matter of the ASIC's interaction with PROC's memory MEM. First we cover the process of fetching data from MEM to ASIC (i.e., a load).

1) Once your ASIC has consumed and understood *cmd.addrR* it should fetch (load) input data from MEM (i.e., fetch $\mathbf{W}_k$ and $\mathbf{x}_k$) and operate on it. Your system gets its data through the `mem.resp` interface. But before you can get any data you have to request it.

2) You make requests for data from memory on the `mem.req` interface. As with the `cmd` and `resp` interfaces the `mem.req` interface also comes with a val/rdy interface to let your ASIC and MEM know that a message has been exchanged. Beyond that, there are a bunch of ways in which you can specify your request on `mem.req`.

3) The simplest way to make a request for data on `mem.req` is to set its `mem.req.addr` field to the address of the byte that you want. The ASIC should also set the `mem.req.cmd` field to 0 to indicate that it is making a request to load data from MEM into your ASIC (as opposed to making a request to store data). For the simplest case, you should also set the `mem.req.typ` field to 0. This indicates that you are requesting only 1 byte of memory. However the interface through which MEM sends actual data to your ASIC is bigger than one byte. Your MEM interface *responds* with data on a 64-bit interface (`mem.resp.data`, more on this soon) so it can actually reply with 8 bytes at-a-time. To initiate such a response you could actually set `mem.req.typ` to 1, 2, or 3 and these will tell MEM to simultaneously return 2, 4, or 8 bytes, respectively from memory. These returned values will be contiguous memory contents. So if your `mem.req.addr` field was 5 and you set `mem.req.typ` to 2 then you are effectively requesting to return 4 bytes in your 64-bit interface with the first byte from address 5, the 2nd from 6, the 3rd from 7, and the 4th from 8.

4) Once a request is sent you can expect to eventually get a response from MEM. You don't know ahead of time when it will come back, but it could be as fast as the next clock cycle after your request is confirmed by the `mem.req` val/rdy interface.

5) The data that you requested will come back on the `mem.resp` interface. Notice from Fig. 1 that this interface only has a val connection for flow control (no rdy signal is used). MEM does not care whether you grab data from it or not (i.e., it has no need for a rdy signal from you). If the ASIC is waiting for some response from MEM you should probably grab what's valid on the `mem.resp` interface.

6) The actual stored data coming back on the `mem.resp` interface appears on the `mem.resp.data` field. This field is 8-bytes wide as implied in the discussion above. And as discussed above, this field can hold up to 8 bytes that you had previously requested. Information about the data in `mem.resp.data` is present in the `mem.resp.addr` and `mem.resp.typ` fields which, respectively, tell your ASIC the starting address of the bytes present in `mem.resp.data` and how many data bytes there are in `mem.resp.data`.

1.4.3 ASIC Stores to MEM

The list above summarized the process of fetching (loading) data from MEM into your ASIC. As long as you follow the protocols outlined above you can grab data any way you wish. And of course your ASIC can work on the data in any way it wishes as well. Now, we can turn our attention to how your ASIC's results (i.e., $\mathbf{r}_k$) are reported back to MEM (i.e., a store).

1) If you understood how `mem.req` described above worked, then you will have very little problem understanding how to send your $\mathbf{r}_k$ to MEM.

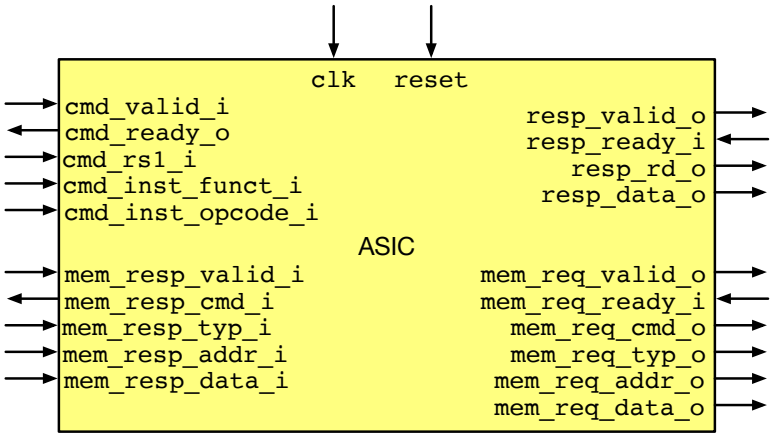

Figure 2: That port names and directions that your asic *must* adhere to.

2) Once you have some bytes of computed data, then to send them back all you have to do is engage with your `mem.req` interface again. In this case you again specify `mem.req.addr` but now you note what the starting address of MEM is for where you want to *store* data into MEM from your ASIC (rather than *load* data into your ASIC as above). Now, you'll have to set `mem.req.cmd` to 1 to tell memory that you want to store data. If you have 1, 2, 4, or 8 bytes that you want to store in one go then you indicate that by setting `mem.req.typ` to 0, 1, 2, or 3, respectively. And that's about it aside from the fact that you should actually place the data you want to send in the `mem.req` interface's `mem.req.data` field. As usual, a val/rdy protocol tells both ASIC and MEM when the exchange is completed.

2 Verification & Simulation

For your design your **you must** use the port names listed (in the "Port name") column of Tables 3, 4, 5, 6. You'll also need a `clk` port (call it exactly that, **not**, for example `Clk`). You'll also need a `reset` port (call it exactly that). In our testbench we'll toggle a port named `reset` at the beginning of the sims (i.e., just before launching *cmd.initialize*). Correctly adhering to these needs will result with an ASIC that presents itself as shown in Fig. 2.

2.1 Functional Verification

Basic verification for this system is pretty simple. For a given PROC command sequence and MEM content we expect a certain byte sequence, representative of the expected **r** (for

possibly multiply k's). After your ASIC signals that it has completed the final load of $\mathbf{r}$ into MEM, the MEM addresses are checked. If all MEM addresses contain the data that they need to contain your system is verified. If they don't, your system fails.

You could imagine a more sophisticated checking scheme which scans memory every time the ASIC writes something new into it. That way you could isolate more easily when an error occurred during the ASIC's process of writing its results to MEM.

We'll need verification across the three main representation: functional RTL, synthesized gate-level, and physically synthesized transistor level. More details to come here. First object of business is to make sure your Verilog works in straight RTL and that it keeps working after logical synthesis with Design Compiler.

2.2 Performance Simulation

We'll also be interested in how fast your system works and how efficiently it was designed (that is how much area it consumes and how much energy it takes to finish a job). We'll figure out some overall metric for you to shoot for here. For now, just make it work correctly.

3 Advice

Draw a beautiful picture of a simple design (you will have to show us your conceptualization in a final report, if your report is ugly and sloppy you'll lose marks). The earlier you can start partitioning your design into smaller parts the better (i.e., you picture can start off at 1 block but hopefully will be expressed as a multiple blocks shortly thereafter). Start coding up your smaller parts and testing them for correct functionality. Test early and often.

Get something working as quickly as possible. Build the simplest implementation you can imagine. Figure out how to run at least some legit tests through it. Don't be afraid to start small (e.g., assuming just $M = 1$ and $N = 1$), but code with scaling in mind (i.e., be aware that you'll probably want to expand your code, so parametrize rather than hard-code as much as you are able).

You can start off assuming that your interfaces are always ready to take your ASIC's outputs and can always provide the next input. But beware, in general, your interfaces may take a number of cycles (unknown how many) before they are ready to take your output and may also have unknown latency in delivering input requests to your ASIC as well. So eventually you'll have to plan for those cases too.

Table 1: Bitfield designation for *cmd.** instructions

31 25	24 20	19 15	14	13	12	11 7	6 0
funct7	rs2	rs1	xd	xs1	xs2	rd	opcode
7	**5**	**5**	**1**	**1**	**1**	**5**	**7**

Table 2: Bitfield settings for *cmd.** instructions

instruction	**funct7**	**rs2**	**rs1**	**xd**	**xs1**	**xs2**	**rd**	**opcode**
cmd.initiate	0x1	X	X	X	X	X	X	*settings*
cmd.size	0x2	X	X	X	X	X	X	X
cmd.addrW	0x4	X	X	X	X	X	X	X
cmd.addrX	0x6	X	X	X	X	X	X	X
cmd.addrR	0x8	X	X	X	X	X	X	X

4 Command and Interface Details

4.1 Command Instruction (cmd)

The *cmd.** signals that PROC sends to the ASIC are 32-b wide and adhere to the bitfield pattern shown in Table 1. The names of the 8 bitfields in a command are given as well as the total number of bits they consume (bottom) and the bit indexes that they span (top). Details about the bit field settings for the different command and response types are as shown in Table 2. The X's in this table mean that you don't care about those contents.

The *settings* in `cmd.opcode` for the *cmd.initiate* instruction mean the following:

1) `7'b000_0000`: $\mathbf{r} = \mathbf{y}'$ 8-b wide
2) `7'b000_0010`: $\mathbf{r} = \mathbf{z}$ 8-b wide
3) `7'b000_0100`: $\mathbf{r} = \phi$ 8-b wide
4) `7'b000_0001`: $\mathbf{r} = \mathbf{y}'$ 16-b wide
5) `7'b000_0011`: $\mathbf{r} = \mathbf{z}$ 16-b wide
6) `7'b000_0101`: $\mathbf{r} = \phi$ 16-b wide

As noted in § 1.4.1 some of the commands are just indications for the ASIC to grab relevant data from the `cmd.rs1` port. Here is how the data is organized on this port for each relevant instruction.

1) For *cmd.initiate* `cmd.rs1[15:0]` holds the value for `a` and `cmd.rs1[31:16]` holds the value for k. Note that the first three bits of k, that is `cmd.rs1[18:16]`,

denote k', a setting denoting how data is distributed in memory. If $k' = 1$ the 8-bit inputs comprising **W** and **r** appear every 64-bits (in a double-word-aligned fashion). That is, if, say, **W**'s starting address is `0x00` and if $k' = 1$ then **W**'s first 8-bit element is at `0x00`, its second 8-bit element is at `0x08`, its third elment is at `0x10`, etc.[1].

On the other hand, if $k' = 2$, it means that 2 8-bit digits appear every 64-bits. So, for example if **W**'s starting address is still `0x00` it means that there are two consecutive eight bit numbers starting at this address, one at byte address `0x00` and the other at byte address `0x01`. If **W** happened to have at least four elements that means that in this example there would also be two numbers at byte addresses `0x08` and `0x09`.

This arrangement continues for k' up to 7. If you wish all 64-bits to be occupied by eight 8-bit numbers set k' to 0. Note that the memory only accepts double-word aligned byte addresses (that is, all address requests to memory must be multiples of 8).

For reference, the `typ` (type) field in the `mem.req` (Table 5) and `mem.resp` (Table 6) interfaces functions just like k' and thus allows instructions to memory to fetch multiple (up to eight) digits at a time.

If the elements that comprise **W** and **r** are 16-b long the same formalism can still be applied, but now the ASIC would have to be aware that it is grabbing 16-bit values. Of course now, the most 16-b numbers that can fit in a 64-b double-word is four.

2) For *cmd.size* `cmd.rs1[15:0]` holds the value for M, `cmd.rs1[31:16]` holds the value for N.

3) For *cmd.addrW* `cmd.rs1[31:0]` hold the starting address for $\mathbf{W}_k$.

4) For *cmd.addrX* `cmd.rs1[31:0]` hold the starting address for $\mathbf{x}_k$.

5) For *cmd.addrR* `cmd.rs1[31:0]` hold the starting address for $\mathbf{r}_k$.

4.2 Response Signal (resp)

.

The ASIC's *resp.data* message to PROC is relatively simple. If the ASIC thinks that is successfully computed all the requested results it should put the value 1 in the *resp.data* message. If your ASIC thinks that it has failed to compute the results it should put the value 2 in the *resp.data* message. For all of these responses, the *resp.data* message should be accompanied by a *resp.rd* message that is always set to 1. After either a successful or failed response from your ASIC has been consumed by PROC, your ASIC should ready itself

[1] Recall, **W** is arranged in row-major order.

Table 3: Command (`cmd`) interface details

Direction	Port name	Default value	Description
output	cmd_ready_o	0	Control lines
input	cmd_valid_i	0	
input [6:0]	cmd_inst_funct_i	funct7	Instructions for ASIC
input [4:0]	cmd_inst_rs2_i	rs2	Source register IDs
input [4:0]	cmd_inst_rs1_i	rs1	
input	cmd_inst_xd_i	xd	Set if destination register exists
input	cmd_inst_xs1_i	xs1	Set if source registers exist
input	cmd_inst_xs2_i	xs2	
input [4:0]	cmd_inst_rd_i	rd	Destination register ID
input [6:0]	cmd_inst_opcode_i	0x1/0x2/0x3/ 0x4	Custom instr opcode, for multi-accel
input [63:0]	cmd_rs1_i	rs1_data	Source register data

Table 4: Response (`resp`) interface details

Direction	Port name	Default value	Description
input	resp_ready_i	0	Control lines
output	resp_valid_o	0	
output[4:0]	resp_rd_o	rd	Destination register ID in the response
output[63:0]	resp_data_o	rd_data	Destination register data in the response

to receive a new *cmd.initiate* instruction. Thus your ASIC can ready itself to potentially receive another stream *k*.

4.3 Interface Ports

The details of the `cmd` interface are tabulated in Table 3.

The details of the `resp` interface are tabulated in Table 4.

The details of the `mem.req` interface are tabulated in Table 5.

The details of the `mem.resp` interface are tabulated in Table 6.

Table 5: Memory request (`mem.req`) interface details

Direction	Port name	Default Value	Description
input	mem_req_ready_i	0	Control lines
output	mem_req_valid_o	0	
output[39:0]	mem_req_addr_o	addr	Mem address corresponds to read/write
output[4:0]	mem_req_cmd_o	cmd	Memory request opcode [0=load, 1=store]
output[2:0]	mem_req_typ_o	typ	Width of resp; [(0,1,2,...,7)→(8,1,2,...,7)B]
output[63:0]	mem_req_data_o	w_data	Store data

Table 6: Memory response (`mem.resp`) interface details

Direction	Port name	Default value	Description
input	mem_resp_valid_i	0	Control
input [39:0]	mem_resp_addr_i	addr	Displays load/store request addr
input [4:0]	mem_resp_cmd_i	cmd	Returns command code of request
input [2:0]	mem_resp_typ_i	typ	Indicates width of the data in response
input [63:0]	mem_resp_data_i	data	Contains data response to a load request